UNIVERSITY OF OSLO
Department of Informatics

Analysis and implementation of high-precision finite element methods for ordinary differential equations with application to the Lorenz system

Master thesis

Benjamin Kehlet

February 5, 2010

Abstract

This paper considers computing accurate solutions on the interval $[0, 1000]$ of ordinary differential equations. This includes implementation of high precision ode solvers and methods to verify the accuracy of the computed solution even for problems with chaotic behaviour. In this paper, we compute an accurate solution of the Lorenz system.

We integrate the **DOLFIN** ODE solver with the GNU Multiple Precision Library (GMP) and are thus able to solve ODEs with arbitrary precision. We extend the ODE solver with general tools for a posteriori error analysis, including solving the linearized dual problem, and storing the primal solution and computing stability factors. In addition, we implement a number of optimizations in **DOLFIN** to make it possible to use methods with high order (~ 200) with the solver.

Using these tools we study the computability of the Lorenz system in detail and show that chaotic dynamical systems, like the Lorenz system, are indeed computable.

Contents

List of Figures

Acknowledgments

I would like to thank everyone at Simula Research Lab and in particular the students at "Drivhuset". Very special thanks to my supervisor Anders Logg for introducing me to a posteriori error analysis and for demonstrating how well academia and open source fit together.

Oslo, february 2010

Benjamin Kehlet

Chapter 1

Introduction

We consider numerical approximations of the initial value problem

$$\begin{cases} \dot{u}(t) &= f(u(t),t) \qquad t \in (0,T] \\ u(0) &= u_0 \end{cases}$$

where u_0 is a given initial condition and $T > 0$ is a given final, error control is of crucial importance. The solution has no value if we cannot guarantee it to be within a chosen tolerance TOL.

1.1 Background

In [14], Edvard N. Lorenz showed how small perturbations of the initial data of non-linear ODEs could blow up. He established the notion of *stability or instability of the trajectories with respect to small modifications.* In the same paper, he also presented the simplest system of ODEs that he was aware of with these properties

$$\begin{cases} \dot{x} &= \sigma(y-x), \\ \dot{y} &= rx - y - xz, \\ \dot{z} &= xy - bz. \end{cases}$$

This system is usually solved with the following values of the constants:

$$\begin{aligned} \sigma = &\quad 10, \\ b = &\quad \tfrac{8}{3}, \\ r = &\quad 28. \end{aligned}$$

We will use the initial data

$$u_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

The system became known as *Lorenz system*. Despite its age, the Lorenz system still holds a prominent position in the literature on what has become known as chaotic dynamical systems.

The Lorenz system has been studied intensively[1], see [19] for a summary. The 14th of the famous "Smale's 18 mathematical problem for the next century" in [17], deals with whether (1.1) is the same attractor as a geometrically defined ODE called "geometric Lorenz attractor". This particular problem was solved by Tucker in [18] in 1998.

Even if many interesting problems can be derived from (1.1), our concern in this paper is the computability, i.e., given a tolerance TOL, can we compute an approximate solution U which satisfies the tolerance. At first sight, the answer seems to be no. The classic method for error control is to give a bound of the error in terms of the maximum norm of the Jacobian of f. However for the Lorenz system this leads to an error estimate which is of order e^{100t}. Due to this, some researchers concluded that it is impossible to compute accurate solutions of chaotic systems, like Lorenz', except for a very small interval.

This picture changed when Don Estep and Claes Johnson developed a technique for estimating the error after a solution is computed – *a posteriori*. This is done in terms of a linearized dual problem. In [4] they introduced the notion of a solution's *computability* given a vector norm, a tolerance TOL and an amount of computational work. They showed that they could compute accurate solutions of Lorenz system up to $T \approx 25$ [FIXME: check] with the computational power of a workstation.

In [10], Logg computes accurate solutions up to $T \approx 40$ using cG(15), but then reaches a limit and concludes

> To get further than $T = 40$ we must thus do something else than decreasing the time-step or increasing the order of the method. We must decrease the computational error, the accumulated round-off error.

So in this work, we take on the challenge of computing accurate solutions of the Lorenz system over time intervals.

1.2 Approach

Our goal is to compute a solution to Lorenz system and then verify that is actually the true solution, i.e., the size of the error is within our chosen tolerance.

Since it seems to be the round-off error that sets the limit, we first need to modify our ODE solver to be able to work with arbitrary (but still finite) precision.

[1] A search for "Lorenz system" at scholar.google.com yields 356 000 results.

We follow the approach used in [12]. This means computing a number of solutions with different methods and orders and then comparing them. Solutions computed with different method should obviously agree. When increasing the order of the method we will expect the solution to be correct on a longer interval.

Our goal is to compute and verify the solution of the Lorenz system for $T = 1000$ with a global tolerance $\mathrm{TOL} = 0.1$.

We then turn our attention to the stability factors of Lorenz system. We compute the stability factors with the techniques developed by Estep and Johnson, i.e., with use of the linearized dual problem. By studying how the stability factors evolve as T increases, we will gain insight in stability properties of (1.1) and confirm what precision is needed to compute accurate solutions longer intervals.

When we have a satisfying solution to (1.1) we use that to study some properties of the error $e = U - u$.

1.3 DOLFIN

We use the ODE Solver in **DOLFIN** [9] as the basis of our implementation. **DOLFIN** is the differential equation solver of the open source software project FEniCS, see [5], which aims at automating the process of computational mathematical modeling, see [8].

DOLFIN implements continuous and discontinuous Galerkin methods for ODEs. In this project, we extend and improve the error analysis capabilities to meet our requirements. These patches have been pushed to the official repository and are now part of the **DOLFIN**ODE solver.

DOLFIN is implemented in C++. All code written in this project, and hence all code in this paper, will therefore also be in C++. All references to **DOLFIN** in this paper will be to the ODE solver part.

1.4 Outline

- We first present the relevant theory including the Galerkin methods and the a posteriori error analysis in Chapter 2.

- Chapter 3 is concerned with **DOLFIN** and the implementation details that made it possible to compute solutions with high precision and high order methods. The chapter includes integration of **DOLFIN** with a bignum library and some new features, necessary for computing solutions of the dual problem. Also, a number of optimizations which made **DOLFIN** able to compute high-precision solutions within reasonable time are described here.

- We are then ready to uncover our actual results in Chapter 4. We present a number of computed solutions of the Lorenz system. We also present computed solutions of the dual problem and the stability factors as function of time. Assuming that we have computed an accurate solution we also compare this solution with solutions computed with lower order to be able to study the error more in detail.

- Finally, in Chapter 5, we discuss the results and whether we can accept them as approximations of the true solution of the Lorenz system. We also mention and compare with some other results on the Lorenz system and their influence on our work.

Chapter 2

Theory

We here present the theory relevant for our work. Our error analysis will be based on connecting the error to the residual through the linearized dual problem.

2.1 Notation

We let I denote the interval $(0, T]$ on which we are computing solutions. We partition I into M sub-intervals such that

$$I = \{I_j\}, \quad j = 1, ..., M$$

We will let $||\cdot||$ denote the euclidean norm. For a discontinous function we will let $[\cdot]$ denote a jump term suct that

$$[f]_j = f(x_j^+) - f(x_{j-1}^-)$$

2.2 The continuous Galerkin method for ODEs

The continuous Galerkin method of order q, cG(q), reads: Find an approximate solution $U \in V$ such that

$$\int_0^T (\dot{U}, v)\ \mathrm{d}t = \int_0^T (f(U, \cdot), v)\,\mathrm{d}t \qquad \forall v \in W \tag{2.1}$$

Here W is the test space and consists of all discontinuous piecewise polynomials of degree $q-1$, or more formally

$$W = \{v : v|I_j \in \mathcal{P}^{q-1}(I_j), j = 1, ..., M\}$$

The trial space, V, is the function space where we seek the solution. For cG(q), the trial space is the space of continuous piecewise polynomials locally of degree q:

$$V = \{v \in [C([0, T])]^N : v \in P^q(I_j), j = 1, ..., M\}.$$

2.3 The discontinuous Galerkin method for ODEs

The discontinuous Galerkin method of order q, dG(q), reads: Find an approximate solution $U \in V$ such that

$$([U]_{j-1}, v(t_{j-1}^+)) + \int_{I_j} (\dot{U}, v)\,\mathrm{d}t = \int_{I_j} (f(U,\cdot), v)\,\mathrm{d}t \tag{2.2}$$

For dG(q) the test space is the same as the trial space.

$$V = W = \{v : v|I_j \in \mathcal{P}^{q-1}(I_j), j = 1, ..., M\}$$

2.4 Nodal basis and quadrature

What remains n order to implement these methods, is to choose basis functions and quadrature. These topics are beyond the scope of this paper. We note that our ODE solver, **DOLFIN**, chooses quadrature such that that nodal basis coinside with the quadrature points. This leads to Lobatto quadrature for the cG(q) method and Radau quadrature for the dG(q) method. For further details, we refer to [11].

2.5 The residual vector

Given an approximate solution U to (1) we define the residual vector as

$$R(U,t) = \dot{U}(t) - f(U(t),t)$$

This applies to both cG(q) and dG(q).

2.6 Orthogonality

From (2.1), it follows that

$$\int_{I_j} R(U,\cdot)v\,\mathrm{d}t = 0$$

while from (2.2 we get

$$([U]_{j-1}, v(t_{j-1}^+)) + \int_{I_j} (R(U,\cdot), v)\,\mathrm{d}t = 0.$$

In this sense the residual vector is orthogonal to the test space. We typically refer to this as the *Galerkin orthogonality*.

2.7 The dual problem

The linearized dual problem is defined as

$$\begin{cases} -\dot{z}(t) & = A^T(t)z(t) \\ z(T) & = z_T \end{cases} \tag{2.3}$$

where

$$A(t) = \int_0^1 f'(su(t) + (1-s)U(t))\, ds$$

We note that the dual problem is itself an IVP on the form (1). However, as the initial data is given at time T, the dual problem runs backward in time. We will address this shortly.

The key property of the dual problem is that it connects the error to the residual, R, of the computed solution. From the dual we will also derive stability factors. Different stability factors measures the growth of different types of error.

2.7.1 The reversed dual problem

Computing solutions of the dual problem directly may be cumbersome as it runs backward in time. To avoid writing a specialized time stepper, our goal is now to rewrite (2.3) to run forward in time. We define

$$w(t) := z(T-t).$$

Then

$$\begin{aligned} w'(t) &= z'(T-t)\cdot(-1) \\ &= A^T(T-t)z(T-t) \\ &= A^T(T-t)w(t) \end{aligned}$$

and

$$w(0) = z(T-0) = z(T)$$

Thus, if we define $B(t) := A^T(T-t)$ and $w_0 := z(T)$, then

$$\begin{cases} w'(t) & = B(t)w(t) \\ w(0) & = w_0 \end{cases} \tag{2.4}$$

Now, a solution $w(t)$ of (2.4) is also a solution $z(T-t)$ of (2.3), but as (2.4) is a normal initial value problem running forward in time, we can use our normal time stepper. We just need to flip the t axis when the solution is computed.

We note that when we in this paper refer to computed solutions of (2.3), it was in fact (2.4) that was fed into our time stepper.

2.7.2 Approximating u

We will use numerically computed solutions of the dual problem to estimate the error. Yet, one problem has arised: We observe that solving (2.3) requires evaluation of u, which we obviously can't... The closest we can get is to use U as an approximation. When computing z we are forced to replace A with $\hat{A}$ defined as

$$\hat{A}(t) = \int_0^1 f'(sU(t) + (1-s)U(t)ds = f'(U(t)),$$

so $\hat{A}$ is the Jacobian of the primal problem.

We will address the consequences of this a little more in Section 5.2.1.

2.7.3 Initial data for the dual

Different choices of data for the dual problem will give different results in the error estimates. The error representation below assumes that

$$z_T = \xi_i$$

where ξ_i denotes the ith unity vector. This allows us to compute componentwise error estimates. We note that, in general, this is expensive as it requires us to do N independent computations of the dual. However, with only 3 components in the Lorenz system, we can afford this.

2.8 Error representation

The error

$$e = U - u$$

is obviously not computable as we don't know u. Still, ensuring that the error is less than our given tolerance TOL is of crucial importance, as the computed solution has no value otherwise.

Using the linearized dual problem, we connect the error to the residual. This is the the most important result in a posteriori analysis and references include [2], [3] and [11].

Theorem 2.8.1. *Let ξ_i denote the ith unitity vector and let e_i denote the ith component of e. Then*

$$e_i(T) = \int_0^T (R(U,\cdot), z)\,\mathrm{d}t$$

Proof. We first note that

$$\begin{aligned}(A(t), e(t)) &= (\int_0^1 \frac{\partial f}{\partial u}(sU(t) + (1-s)u(t))\, ds, e(t)) \\ &= \int_0^1 \frac{\partial f}{\partial s}(sU(t) + (1-s)u(t))\, ds \\ &= f(U(t)) - f(u(t))\end{aligned}$$

Observing that $-\dot{z} - A^T z = 0$ due to the definition of the dual, we start with

$$\begin{aligned}0 &= \int_0^T (e, -\dot{z} - A^T z)\, \mathrm{d}t \\ &= \int_0^T -(e, \dot{z})\, \mathrm{d}t - \int_0^T (e, -A^T z)\, \mathrm{d}t \\ &= \int_0^T -(e, \dot{z})\, \mathrm{d}t - \int_0^T (Ae, z)\, \mathrm{d}t. \end{aligned} \tag{2.5}$$

Using partial integration we get

$$\begin{aligned}\int_0^T -(e, \dot{z})\, \mathrm{d}t &= -(e(T), z(T)) + (e(0), z(0)) - \int_0^T (\dot{e}, z)\, \mathrm{d}t \\ &= -e_i(T) + \int_0^T (\dot{e}, z) \mathrm{d}t\end{aligned}$$

where we used that $z(T) = \xi_i$ and assumed $e(0) = 0$ (i.e. not taking error in the initial data into account).

Combining this with 2.5, we get

$$\begin{aligned}e_i(T) &= -\int_0^T (Ae, z) + \int_0^T (\dot{e}, z)\, \mathrm{d}t \\ &= \int_0^T (f(u) - f(U), z) + (\dot{e}, z)\, \mathrm{d}t \\ &= \int_0^T (f(u) - f(U) - \dot{u} + \dot{U}, z)\, \mathrm{d}t \\ &= \int_0^T (R(U), z)\, \mathrm{d}t,\end{aligned}$$

which completes the proof. □

Next we will refine our error representation slightly. Our error representation consists of two parts

$$E = E_G + E_C,$$

where E_G represents the error introduced by the Galerkin method, while E_C represents the round-off error which is unavoidable since we are using computer arithmetic with finite precision.

To achieve an error representation of this form we will add and subtract the term $\int_0^T (R, \pi_k z)$ where $\pi_k z$ is some test space interpolant of z. We end up with

$$E = \int_0^T (R, z - \pi_k z)\mathrm{d}t + \int_0^T (R, \pi_k z)$$

which leaves the quantity E unchanged.

2.8.1 Galerkin error

We now consider only errors introduced by the Galerkin method. Our error estimate has the form

$$E_G = \int_0^T (R, z - \pi_k z)\,\mathrm{d}t \leq \int_0^T ||R|| \cdot ||z - \pi_k z||\mathrm{d}t$$

We now focus on the quantity $||z(t) - \pi_k z(t)||$ with $t \in [t_{m-1}, t_m]$ to obtain a computable expression. Choosing $\pi_k z(t)$ to be the qth order Taylor expansion of z around $t_0 = \frac{t_m + t_{m-1}}{2}$, we can rewrite as the remainder in integral form

$$\begin{aligned}
||z(t) - \pi_k z(t)|| &= ||\frac{1}{q!}\int_{t_0}^{t} z^{(q+1)}(y)(y - t_0)^q\,dy|| \\
&= \frac{1}{q!}\int_{\frac{t_m+t_{m-1}}{2}}^{t} z^{(q+1)}(y)(y - \frac{t_m + t_{m-1}}{2})^q\,dy|| \\
&\leq \frac{1}{q!}\int_{\frac{t_m+t_{m-1}}{2}}^{t} ||z^{(q+1)}(y)\left(\frac{k}{2}\right)^q||\,dy \\
&= \frac{1}{2^q q!}k^q\int_{\frac{t_m+t_{m-1}}{2}}^{t_m} ||z^{(q+1)}(y)||\,dy \\
&\leq \frac{1}{2^q q!}k^q\int_{t_{m-1}}^{t_m} ||z^{(q+1)}(y)||\,dy \\
&= C_q \cdot k_m^q ||z^{(q)}(t)||
\end{aligned}$$

where $C_q = \frac{1}{2^q q!}$.

Using this we get

$$\begin{aligned}
\int_0^T ||R|| \cdot ||z - \pi_k z|| \, \mathrm{d}t &\leq \sum_{m=1}^{M} \int_{t_{m-1}}^{t_m} ||R|| \cdot C_q \cdot k_m^q ||z^{(q)}(t)|| \, \mathrm{d}t \\
&\leq \sum_{m=1}^{M} C_q k_m^q \max_{t \in I_m} ||R|| \cdot \int_{t_{m-1}}^{t_m} ||z^{(q)}(t)|| \, \mathrm{d}t \\
&\leq \max C_q k_m^q ||R|| \cdot \int_0^T ||z^{(q)}|| \, \mathrm{d}t \\
&= \max C_q k_m^q ||R|| \cdot S_G(T)
\end{aligned}$$

We define the stability factor with respect to the error introduced by the Galerkin method as

$$S_G(T) = \int_0^T ||z^{(q)}(t)|| \mathrm{d}t$$

2.8.2 Round-off error

So far the analysis has relied on the Galerkin orthogonality, i.e. the term $\int_0^T (R(U), \pi_k z)$ being zero. However when using finite precision arithmetic we are not able to solve the discrete equations exactly, since we must expect all floating point operations to introduce a round-off error of order ϵ. Taking these errors into account $\int_0^T (R(U), \pi_k z)$ is no longer zero, but represents the computational error.

$$E_C = \left| \int_0^T (R, \pi_k z) \mathrm{d}t \right|$$

To obtain something computable we define the stability factor with respect to computation errors as

$$S_C = \int_0^T |z| \, \mathrm{d}t$$

and the *discrete* residual as

$$\bar{R}_j = \frac{1}{k_j} \int_{I_j} R(U, \cdot) \, \mathrm{d}t$$

We get

$$\begin{aligned} E_C &= \left| \int_0^T (R(U,\cdot), \pi_k z)\,\mathrm{d}t \right| \\ &\leq \sum_{j=1}^{M} \left| \int_{I_j} R(U,\cdot)\pi_k z\,\mathrm{d}t \right| \\ &\approx \sum_{j=1}^{M} k_j \left| \bar{z_j} \frac{1}{k_j} \int_{I_j} R(U,\cdot)\,\mathrm{d}t \right| \\ &= \sum_{j=1}^{M} k_j |\bar{z}|\,|\bar{R_j}| \\ &\leq S_C \max_j \bar{R}_j \end{aligned}$$

where $\bar{z_j}$ is some constant approximation of z over the interval I_j.
We now carry on to obtain a simple estimate of the discrete residual.

$$\begin{aligned} \bar{R}_m &= \frac{1}{k_m} \int_{t_{m-1}}^{t_m} R(U,\cdot)\,\mathrm{d}t \\ &= \frac{1}{k_m}(U(t_m) - U(t_{m-1}) - \int_{t_{m-1}}^{t_m} f(U,\cdot)\,\mathrm{d}t) \\ &\approx \frac{\epsilon}{k_m} \end{aligned}$$

So our final estimate of the computational error is

$$E_C = S_C \max_j \bar{R}_j \approx S_C \frac{\epsilon}{\min k_m} \tag{2.6}$$

Chapter 3

Implementation

In this section we describe how **DOLFIN** were modified to be able to solve ODEs with high precision. Also some new features was implemented to be able to do the error analysis, including computation of the dual problem and storage of the primal solution.

3.1 Arithmetic precision

While $\mathbb{R}$ has infinitely many numbers on any interval, a computer can only distinguish between a finite set of numbers. We denote the set of floating-point numbers $\mathbb{R}_C$. Other numbers must be rounded to the (hopefully) closest number in this set.

We denote the precision in terms of ϵ, defined as

$$\epsilon := \min x : 1 + x \neq 1, \qquad \forall x \in \mathbb{R}_C$$

We will let ϵ_M denote the precision when using normal hardware precision (type `double` in the code). Typically $\epsilon_M \sim 10^{-16}$.

3.2 Bignum library

To compute solutions with extended precision, we need to replace the use of double primitives with a software defined type which supports arbitrary precision — a so called bignum library. This is a classic example of the power of operator overloading in C++. Ideally, we should need only to replace all occurances of `double` with this chosen type, conveniently done with a simple `typedef`.

A number of bignum libraries, both commercial and free, are available. We chose *GNU Multiple Precision library (GMP)* [6] mainly because it is open source (as **DOLFIN**), known to be fast and stable and the documentation is good. It is written in C, but a C++–wrapper which implements

operator overloading is included as part of the package. It later turned out that an extension of GMP named *GNU Multiple Precision Floating Point Library with reliable rounding (MPFR)* [15] is even more specialized on these types of computations. However, it does not include an official C++-wrapper and *GMP* was satisfying for our needs.

Changing to GMP types was straightforward. However, the **DOLFIN** ODE Solver relies on uBLAS in Boost for doing linear algebra. uBLAS is templated so it should be possible for it to use GMP types. However, it turned out that uBLAS relies on non-templated code, which can be used only with primitive types. The entire interface of the ODE Solver had to be changed from using uBLAS types to simple arrays of `real`s, where `real` is a typedef for the GMP type when using extended precision.

3.3 Solving systems of linear equations

We have to be able to solve systems of linear equations $Ax = b$. Unfortunately none of the available solvers could be used with GMP types as they all relied on non-templated code. Instead, we implemented a Gauss–Seidel solver. However, to ensure fast convergence we first compute the inverse of the matrix with the double precision LU Solver for uBLAS, and then use this as preconditioner for our Gauss-Seidel solver. The uBLAS solver is fast, and when preconditioning with this matrix we get a coefficient matrix with condition number $\kappa(A) \approx 1 + \epsilon_M$. This is verified as our Gauss-Seidel solver gains approximately 16 decimal digits of precision per iteration.

3.4 Storing the primal solution

The ODE solver needs to be able to evaluate the computed solution U to solve the dual problem. In order to achieve this a class `ODESolution` was implemented. The main function in the public interface is

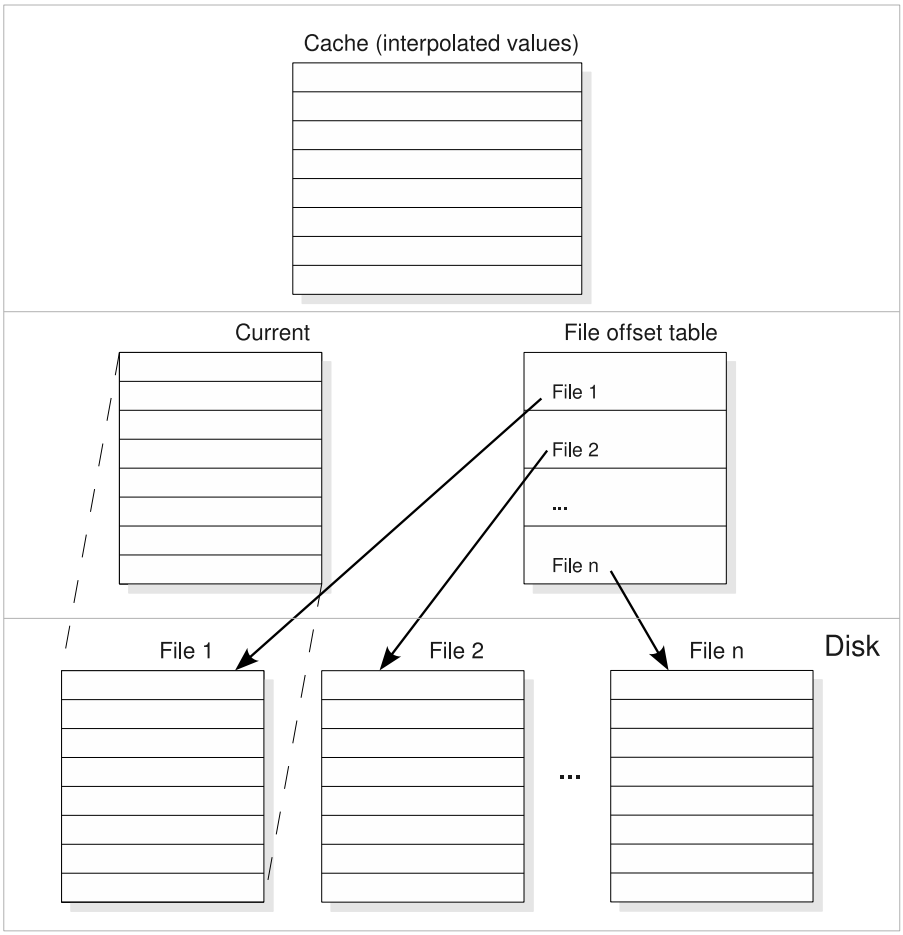
```
void eval(const real& t, real* y)
```

which evaluates `U` at time `t` and places the result in the vector `y`.

3.4.1 Three level storage

The `ODESolution` class needs to store the entire solution. To be able to interpolate the solution in an arbitrary point, it must store all nodal values. This means that the amount of data is of order $\mathcal{O}(N \cdot q \cdot d)$ where d is the number of digits per floating-point number (the precision). We end up with large amounts of data, so a disk based solution is needed. Unfortunately, GMP does not support saving operands on disk, so we have to dump

Figure 3.1: `ODESolution` storage model



them in ASCII format, which requires even more space. When solving the Lorenz system with cG(100), 400 decimal digits of precision and a fixed time step of length 0.05, we end up with approximately 30 GB of data. (Saving operands on disk is planned in GMP in the near future, and `ODESolution` must definitely be updated to reflect this, as it will give a very noticeable speedup).

When solving the dual, the solver repeatedly requests evaluation of the same value. To avoid a full lookup and interpolation, q interpolated values are stored in a cache.

Since ODESolution needs to handle rather large amounts of data, we choose to split up the storage in multiple files. Tens of gigabytes in one single file might be in conflict with the maximum file size of certain file

systems, and it makes the implementation easier and less error prone, not having to deal with parts of files. At all time our implementation keeps the data of one file in memory. The data is stored in a `std::vector`, and we use `std::lower_bound()` to search in memory. `std::lower_bound()` makes a binary search, and has complexity $\mathcal{O}(\log n)$. If the requested `t` is outside the range of what is kept in memory, a search is done in a table which keeps track of which interval is kept in what file and the entire file is read into memory. This has the potential drawback, that if different `t` values are requested from different files, we may end up reading big files into memory on every request. However, when our implementation is used to solve the dual, and when the last q interpolated values are cached, the requests that misses the cache are monotonically decreasing, so in our use this isn't a problem.

The model storage model of the ODESolution class is illustrated in Figure 3.1.

3.4.2 Iterating through a solution

When computing the stability factors as function of time $S(T)$, we need to do one evaulation of U per timestep of S. By choosing the same time step for S as for U, we can use the stored nodal values directly, and thereby avoid the cost of searching and interpolating.

The C++ Standard Template Library (STL) uses a special type of objects called *iterators* when iterating through data structures. We implemented an iterator for ODESolution. The result is an interface which should be familiar to C++–programmers:

```
for (ODESolution::iterator it = u.begin(); it != u.end(); it++)
{
  ODESolutionData& timestep = *it;
  //Do something with timestep
}
```

The class `ODESolutionData` is a simple data structure holding the start time of the time step, the length of the time step and the nodal values.

3.5 Verifying the solver with extended precision

After having replaced the `doubles` with the GMP type and worked around the trouble with the linear algebra back ends, we need to verify that our solver actually works, i.e., that we are able to solve equations with higher precision. We used the well-known *harmonic oscillator*

$$\begin{bmatrix} \dot{u_1} \\ \dot{u_2} \end{bmatrix} = \begin{bmatrix} u_2 \\ -u_1 \end{bmatrix} \tag{3.1}$$

with initial data $u(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The exact solution is $u(t) = \begin{bmatrix} \sin t \\ \cos t \end{bmatrix}$.

We solve (3.1) with $T = 4\pi$ and expect the solution at T to approach $\begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$ as we increase the precision.

Table 3.1: Error convergence as precision is increased. All results are computed with cG(15).

ϵ	Bits per number	error
$8.67 \cdot 10^{-20}$	64	$2.91 \cdot 10^{-18}$
$2.02 \cdot 10^{-29}$	96	$8.76 \cdot 10^{-28}$
$1.09 \cdot 10^{-48}$	160	$3.08 \cdot 10^{-47}$
$3.22 \cdot 10^{-87}$	288	$1.05 \cdot 10^{-85}$
$2.78 \cdot 10^{-164}$	544	$9.85 \cdot 10^{-102}$

The results listed in Table 3.1 show that we get the expected decrease in error. When running the same program with hardware precision the error stops at $e \approx \epsilon_M$. Note that the error follows ϵ quite closely except for the last run, which indicates that we have reached the limit for how precise the method (in this case cG(15)) is able to solve the equation with the chosen time step.

In Figures 3.2 and 3.3, we plot the solutions of 3.1 and the corresponding dual problem.

3.6 Some optimizations in DOLFIN

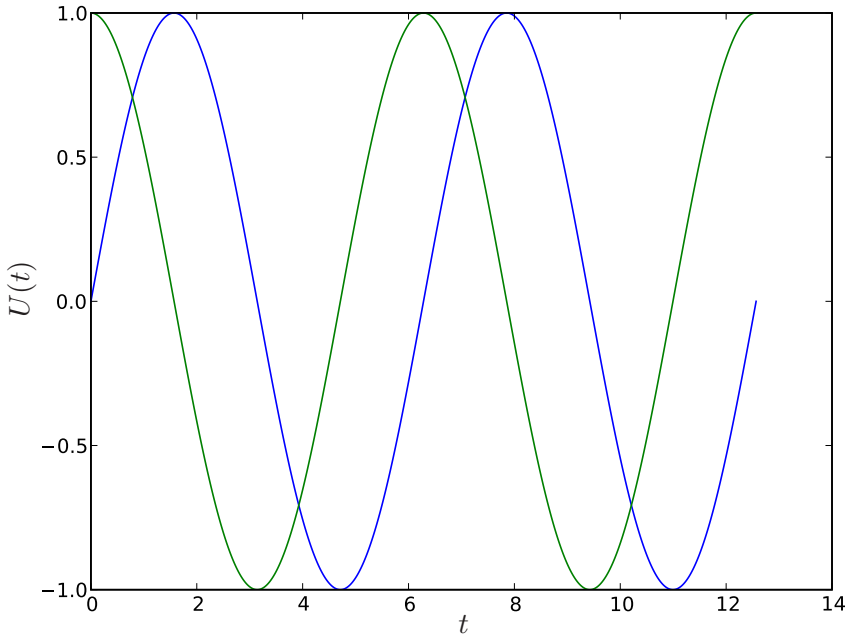
During the work, the ODE solver in **DOLFIN** was challenged more than it had ever been before. We wanted to use very high order methods compared to what had been used. A number of optimizations were made, especially in the initialization phase where the quadrature and nodal weights are computed. Here we describe the most important of them briefly

3.6.1 Computing derivatives of Lagrange polynomials

DOLFIN uses Lagrange polynomials as a basis for the test and trial spaces due to their good numerical properties. The jth Lagrange base is given by

$$L_j(x) = \prod_{i=1, i \neq j}^{n} \frac{x - x_i}{x_j - x_i}$$

Figure 3.2: The Harmonic Oscillator



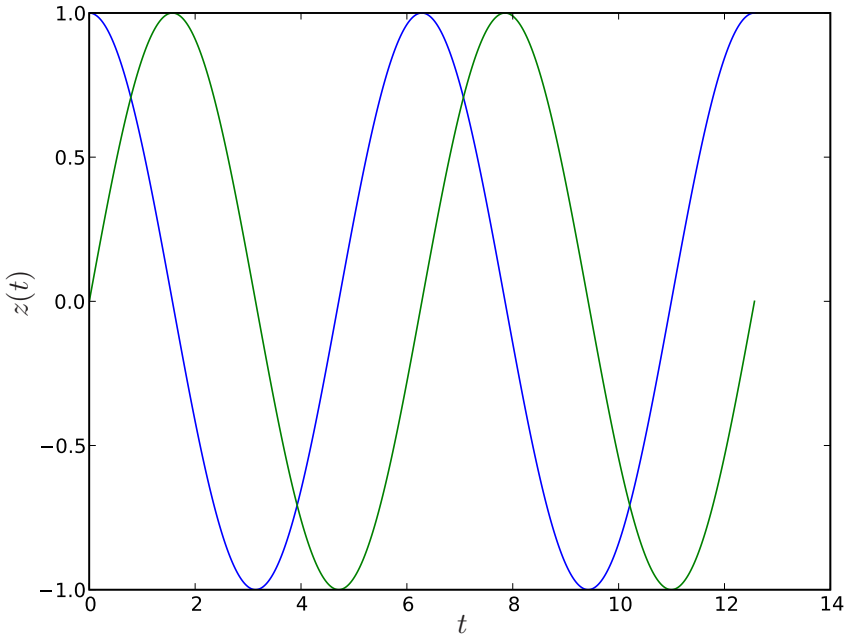
and its derivative is given by

$$L_j'(x) = \sum_{k=1,k\neq j}^{n} \left(\frac{1}{x_j - x_k} \prod_{i=1,i\neq j,i\neq k}^{n} \frac{x - x_i}{x_j - x_i} \right).$$

Naively implemented, this is $O(n^2)$. However, observing that the two loops are almost equal, we can do the following to implement this with linear complexity.

In the case where $x \neq x_l$ for $l = 1, \ldots, n$ and $l \neq j$ we can multiply by

Figure 3.3: The dual of the Harmonic Oscillator



$\frac{x-x_k}{x-x_k}$ and obtain

$$\begin{aligned} L'_j(x) &= \sum_{k=1,k\neq j}^{n} \left(\frac{1}{x_j - x_k} \frac{x - x_k}{x - x_k} \prod_{i=1,i\neq j,i\neq k}^{n} \frac{x - x_i}{x_j - x_i} \right) \\ &= \sum_{k=1,k\neq j}^{n} \left(\frac{1}{x - x_k} \prod_{i=1,i\neq j}^{n} \frac{x - x_i}{x_j - x_i} \right) \\ &= \left(\sum_{k=1,k\neq j}^{n} \frac{1}{x - x_k} \right) \left(\prod_{i=1,i\neq j}^{n} \frac{x - x_i}{x_j - x_i} \right), \end{aligned}$$

which can easily be implemented with one loop and linear complexity.

In the case where $x = x_l$ for $l \in \{1, \ldots, j-1, j+1, \ldots, n\}$, we observe that the product $\prod_{i=1,i\neq j,i\neq k}^{n} \frac{x-x_i}{x_j-x_i}$ is zero, except when $l = k$. This means the formula reduces to

$$L'_j(x) = \frac{1}{x_j - x_l} \prod_{i=1,i\neq j,i\neq l}^{n} \frac{x - x_i}{x_j - x_i},$$

which is obviously of linear complexity.

This is implemented as follows in C++:

```
real Lagrange::ddx(uint i, real x)
{
  dolfin_assert(i <= q);


  real s(0.0);
  real prod(1.0);
  bool x_equals_point = false;

  for (uint j = 0; j < n; ++j)
  {
    if (j != i)
    {
      real t = x - points[j];
      if (abs(t) < real_epsilon())
      {
        x_equals_point = true;
      } else
      {
        s += 1/t;
        prod *= t;
      }
    }
  }

  if (x_equals_point) return prod*constants[i];
  else                return prod*constants[i]*s;
}
```

Note that `constants[i]` is the precomputed product $\prod_{j,i\neq j}^{n} \frac{1}{x_j - x_i}$.

3.6.2 Computing Legendre polynomials

DOLFIN uses Legendre polynomials to compute the basis of the Radau quadrature and also to check that quadrature weights are correctly computed, as the integral should be 2.0 for all orders greater than 0. The $(n+1)$th Legendre polynomial is defined by the recurrence relation

$$P_{n+1}(x) = \frac{(2n+1)xP_n(x) - nP_{n-1}(x)}{n+1}$$

given that $P_0(x) = 1$ and $P_1(x) = x$.

A naive recursive implementation will repeatedly compute the same evaluations a lot of times. Instead, a loop running from 0 to n and just saving the two last values will have linear complexity. However, since the ODE solver requires evaluation at the same x value but different orders a lot of times, caching the computed values will save us even more time, while the cost in terms of memory usage is reasonable. The implementation is as follows:

```
real Legendre::eval(uint nn, real x)
```

```
{
  //check cache
  if (x != cache_x)
  {
    cache[0] = 1.0;
    cache[1] = x;

    for (uint i = 2; i <= n; ++i)
    {
      real ii(i);
      cache[i] = ( (2.0*ii-1.0)*x*cache[i-1] - (ii-1.0)*cache[i-2] )↩
          / ii;
    }
    cache_x = x;
  }
  return cache[nn];
}
```

3.6.3 Computing nodal weights

For the Galerkin methods cG(q) and dG(q), we compute the nodal weights by solving a system of linear equations. When computing the coefficients for this system, we end up with a triply nested loop with complexity $\mathcal{O}(n^3)$. Inside this loop we need to evaluate the test functions and the derivative of the trial functions which is of linear complexity. However, since the evaluation depends only on which quadrature point we are evaluating at and which Lagrange polynomial, we can precompute these values. The cost in terms of memory usage is of order n^2 and the complexity is reduced from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$. The implementation is as follows:

```
real A_real[q*q];

real trial_ddx[nn * nq];
real test_eval[nn * nq];

// evaluate the test functions and the derivative og the trial
// functions at each nodal point and ofr each degree and store
for (uint a = 0; a < nn; ++a) {
  for (uint b = 0; b < nq; ++b) {
    trial_ddx[a*nq + b] = trial->ddx(a+1, qpoints[b]);
    test_eval[a*nq + b] = test->eval(a, qpoints[b]);
  }
}

// Compute matrix coefficients
for (uint i = 0; i < nn; i++)
{
  for (uint j = 0; j < nn; j++)
  {
    // Use Lobatto quadrature which is exact for the order we need, ↩
        2q-1
    real integral = 0.0;
    for (uint k = 0; k < nq; k++)
    {
      //real x = qpoints[k];
```

```
      //integral += qweights[k] * trial->ddx(j + 1, x) * test->eval(↩
          i, x);
      integral += qweights[k] * trial_ddx[j*nq + k] * test_eval[i*nq↩
          + k];
    }

    A_real[i*q+j] = integral;
    _A(i, j) = to_double(integral);

  }
}
```

The dG(q) method was optimized accordingly.

3.6.4 Parallelizing the time slab solver

The ODE Solver has to solve a non-linear equation in each time step. We are using the fixed point iteration method. This piece of code which takes place within an iteration is fairly easy to parallelize:

```
// Update the values at each stage
for (uint n = 0; n < method.nsize(); n++)
{
  const uint noffset = n * ts.N;

  // Reset values to initial data
  for (uint i = 0; i < ts.N; i++)
    ts.x[noffset + i] += alpha*(ts.u0[i] - ts.x[noffset+i]);

  // Add weights of right-hand side
  for (uint m = 0; m < method.qsize(); m++)
  {
    const real tmp = k * method.nweight(n, m);
    const uint moffset = m * ts.N;
    for (uint i = 0; i < ts.N; i++)
      ts.x[noffset + i] += alpha*tmp*ts.fq[moffset + i];
  }
}
```

It updates each entry in x which represents the degrees of freedom of the solution. The size of x is $q \cdot N$ for cG(q) and $(q+1) \cdot N$ for dG(q). Since this takes place in each iteration on each time step, the overhead of creating the threads is significant. We implemented a simple thread pool where each thread is waiting for a mutex and runs one iteration when the mutex is unlocked. Still, it turns out there isn't much to gain when solving with a low value of q and with normal hardware precision. However, when solving with cG(100) and 350 decimal digits of precision, the time was reduced by approximately a factor of 6 when then iteration was parallelized on 8 cores.

3.6.5 Outcome of optimizations

These optimizations have been crucial. Some informal measures of the time used for initialization (before the time stepping starts) before these opti-

mizations are listed in Table 3.2. They are all computed using 200 decimal digits precision and on the same hardware.

Table 3.2: Initialization time prior to optimizations.

Method	seconds	time increase factor
cG((15)	60	-
cG((16)	160	2.67
cG((17)	440	2.75
cG((18)	1208	2.74
cG((19)	3302	2.73

As we can see from table 3.2 the time is clearly exponential and can be approximated by

$$t(q) \approx 2.0 \cdot 10^{-5} \cdot 2.7^q$$

This means that $t(25) \approx 1.2 \cdot 10^6$ seconds or about 14 days, while $t(100) \approx 2.7 \cdot 10^{38}$ or about 10^{31} years!

With the optimizations described above the initialization of cG(100) takes about 102 seconds on the same hardware. This is a speedup of a factor 10^{29}.

3.7 Computing stability factors as function of time

Naively implemented, this involves solving (and integrating) the solution of the dual problem once for each timestep of S. This is obviously not feasible. Fortunately, the dual problem is linear, which we can take advantage of.

In general, if A is a constant coefficient matrix, the initial value problem

$$\begin{cases} \dot{w}(t) &= Aw(t), \\ w(0) &= w_0, \end{cases} \tag{3.2}$$

has the solution

$$w(t) = e^{A \cdot t} w_0,$$

where $e^{A \cdot t}$ denotes the matrix exponential.

When A is not constant we can still approximate the solution by discretizing the problem, since A is almost constant on small intervals:

$$w(t_n) \approx e^{A(t_n)h_n} w(t_{n-1})$$

where $h_n = t_n - t_{n-1}$. The approximate solution is now given by

$$w(t) \approx \prod_{i=1}^{m} \left(e^{A(t_{m-i})h_{m-i}} \right) w_0,$$

where m is the number of discrete timesteps.

Different stability factors reflects stability with respect to different aspects of the computation. For S_C we need to evaluate the qth derivative and w. Since the derivative is given in the definition (3.2), the double derivative is given by the product rule of derivatives:

$$\begin{aligned} \ddot{w} &= \dot{A} \cdot w + A \cdot \dot{w} \\ &= \dot{A} \cdot w + A(Aw) \\ &\approx 0 \cdot w + A(Aw) \\ &= A^2 \cdot w \end{aligned}$$

where we used that $\dot{A}$ is close to constant in each time step. This means that the qth derivative is given by

$$w^{(q)}(t) \approx (A(t))^q w(t)$$

Since we don't need exact values of S we will use the simplest possible approach when evaluating the integral:

$$\begin{aligned} S_q(t) &= \int_0^t ||w^{(q)}(t)|| \mathrm{d}t \\ &\approx \sum_{n=1}^{m} h_n ||w^{(q)}(t_n)|| \end{aligned}$$

We can now compute the stability factor as

$$S_q(t) \approx \sum_{n=1}^{m} \left(h_n \cdot A(t_n)^q \prod_{i=1}^{m} \left(e^{A(t_{m-i}) h_{m-i}} \right) w_0 \right) \tag{3.3}$$

Now, to take advantage of the linearity of the problem, we note that for timestep m, the computation has the form

$$S_q(t) = \sum_{n=1}^{m} \left(K_n \prod_{i=1}^{m} L_i \cdot \boldsymbol{w_t} \right)$$

where K_n and L_i are matrices. The products $\prod_{i=1}^{m} L_i$ are almost the same, except for each time step m is increased by one, so the product consists of one more matrix. Therefore we can reuse the product from the previous timestep, and need only to perform one matrix multiplication to compute the L product. Also note that in the matrix product in (3.3) the matrices are multiplied opposite of the indices, which implies that for each new timestep $S(t_m)$ we must *right-multiply* the product from $S(t_{m-1})$ with L_m.

Unfortunately all matrices here are dense and of size $(N \times N)$ where N is the number of components in the primal problem. This means that computing S is expensive in terms of both memory usage and computation, for all but very small systems of ODEs.

3.7.1 Computing the matrix exponential

Computing the stability factor S_q in (3.3) requires evaluation of the matrix exponential e^B. The matrix exponential is a generalization of the scalar exponential given by

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}$$

which is guaranteed to converge for square matrices. However, much more sophisticated algorithms are available. We have implemented the algorithm described in [16] which uses Padé approximation.

Chapter 4

Results

In this chapter we will present different computed solutions of the Lorenz system, the dual problem and the stability factors.

4.1 Solutions of the Lorenz system

We have computed solutions of the Lorenz system with cG(10), cG(20), ..., cG(100) and dG(10), dG(20), ..., dG(90). We will now compare these solutions. When the length of the timestep k is fixed, we expect the error to decrease as we increase the order q of the method.

In all the computations the global tolerance was set to 0.1. The timestep length is 0.005. The discrete tolerance was set to 10^{-400}. To obtain this precision, each floating-point number required 1344 bits of memory. It took almost 134 hours to compute the solution on an eight-core computer.

In Table 4.1 and Table 4.2 respectively we have listed the t values for which the solutions agree for cG(q) and dG(q) respectively. In Figure 4.1 and Figure 4.2 we have plotted the relevant parts of the solutions. The parts where the solutions agree are plotted with a green solid line, while a blue dotted line indicates that the compared solutions don't agree, i.e. the solution is not accurate. We observe that that interval on which the solution is accurate within the tolerance, increases by approximately 100 when we increase the order by 10. When comparing cG(99) and cG(100) we see that the solutions agree up to 1016.0.

Using cG(100) our computed solution with $u(0) = (1, 0, 0)$ at $t = 1000$ is

$$u(1000.0) = \begin{bmatrix} 12.5 \\ 12.8 \\ 31.9 \end{bmatrix}.$$

We note that our initial computation was done with a precision of 350 decimal digits (and the discrete tolerance set accordingly to 10^{-350}). This

turned out not to be enough to reach $t = 1000$ with the same solution for cG(99) and cG(100).

In Figures 4.3 we have plotted the solution up to $t = 1000$. Figure 4.4 shows the same solution, but now plotted as trajectory in phase space.

Table 4.1: cG(q): Interval on which the solutions agree

Methods	t
cG(10) - cG(20)	105.1
cG(20) - cG(30)	205.8
cG(30) - cG(40)	307.6
cG(40) - cG(50)	407.1
cG(50) - cG(60)	509.1
cG(60) - cG(70)	611.3
cG(70) - cG(80)	716.8
cG(80) - cG(90)	817.0
cG(90) - cG(99)	923.2
cG(99) - cG(100)	1016.0

Table 4.2: dG(q): Interval on which the solutions agree

Methods	t
dG(10) - dG(20)	106.8
dG(20) - dG(30)	209.5
dG(30) - dG(40)	310.5
dG(40) - dG(50)	408.7
dG(50) - dG(60)	510.8
dG(60) - dG(70)	614.9
dG(70) - dG(80)	717.1
dG(80) - dG(89)	817.2
dG(89) - dG(90)	915.0

4.2 The dual problem

We have computed a numerical solution of (2.3). The solution is computed with $q = 50$, $k = 0.05$ and $T = 1000.0$.

The solution oscillates and the extreme values grow with exponential rate as t approaches 0. In Figure 4.6, we have plotted parts of the computed solution, $t \in [0, 25]$ with linearly scaled axes.

In Figure 4.5, we have plotted the absolute values of the same solution of the dual problem with a logarithmic scale on the y axis.

4.3 The stability factors

In Figure 4.8, the computational stability factor is plotted. In Figure 4.9, we have plotted the first component of the same stability factor on the interval $[0, 100]$ in order to see more of the details.

The stability factors with respect to the Galerkin error, S_G, are plotted in Figure 4.7.

4.4 The error

We will now assume that the solution computed with cG(100) is accurate. We refer to 5.1 for a discussion of that. This accurate solution gives us the opportunity to study the error in the lower order methods more in detail.

In Figure 4.10 we have plotted the accurate solution and an approximation computed with cG(1) with $k = 0.05$. We can see how both solutions starts at the same point, the initial position $[1, 0, 0]$. The distance between the two solutions increases and eventually the solution computed with cG(1) departs rapidly from the exact solution.

In Figure 4.11, we have plotted the error for different k. Each point represents a solution computed with cG(1) and with $T = 30$.

We first note that the error is bounded. This makes sense when compared with Figure 4.10, since both the accurate and the "wrong" solution orbit around the two fixed points. As the length of the time step decreases (we move left on the plot), the error also decreases as expected. However, at some point when $k = 10^{-6.5}$ the line becomes choppy and there is a tendency that the error increases as the k decreases. We have come to the point where the computational error dominates.

We obtain the following model of the error:

$$e(k) \approx C_Q k^{p_Q} \cdot C_G k^{p_G} \approx 10^{-8} \cdot k^{-0.45} + 10^{9} \cdot k^{1.97}. \tag{4.1}$$

This model is plotted as the red line in Figure 4.11.

Figure 4.1: Comparing solutions computed with the continuous Galerkin method.

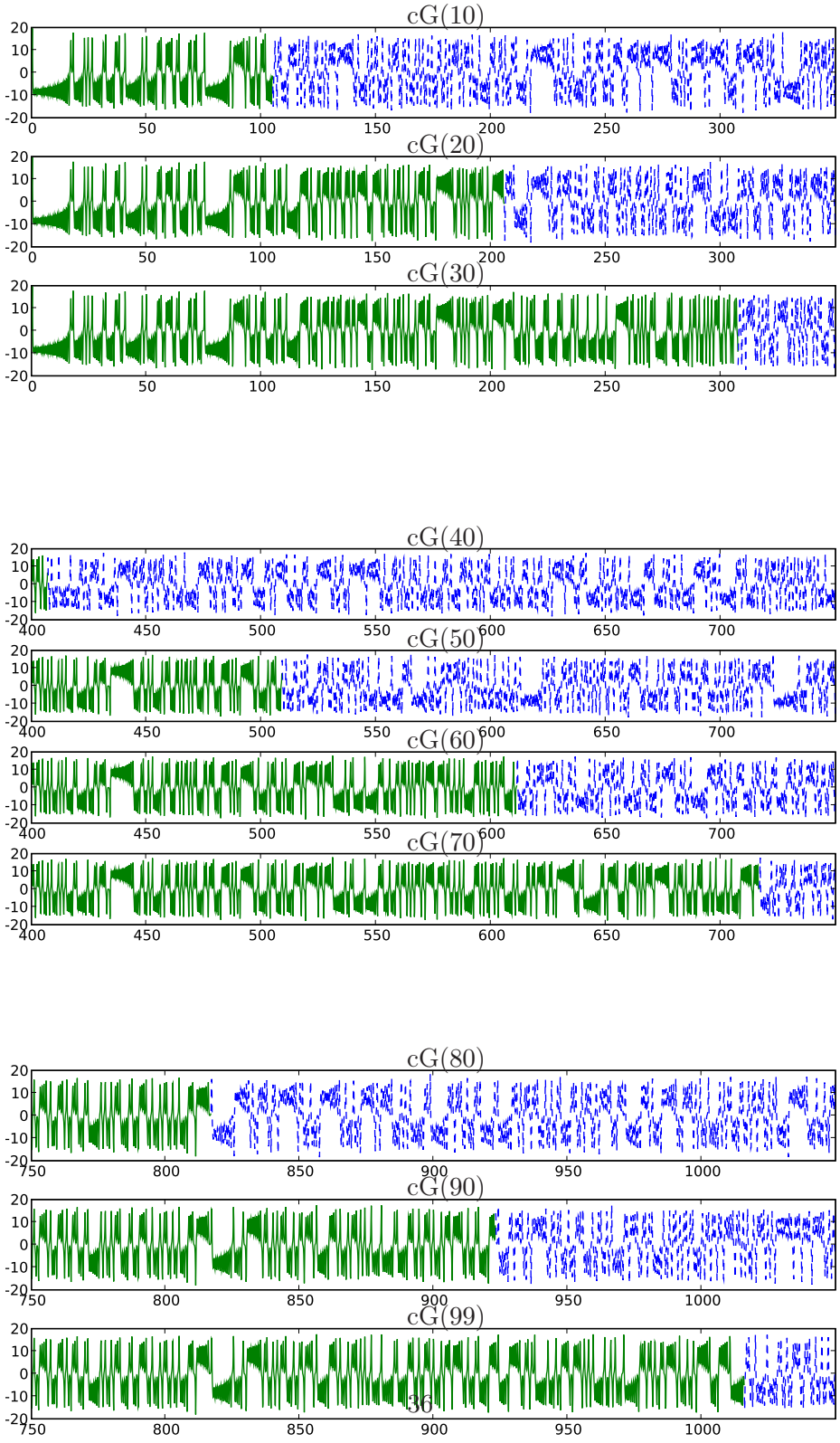


Figure 4.2: Comparing solutions computed with the discontinuous Galerkin method.

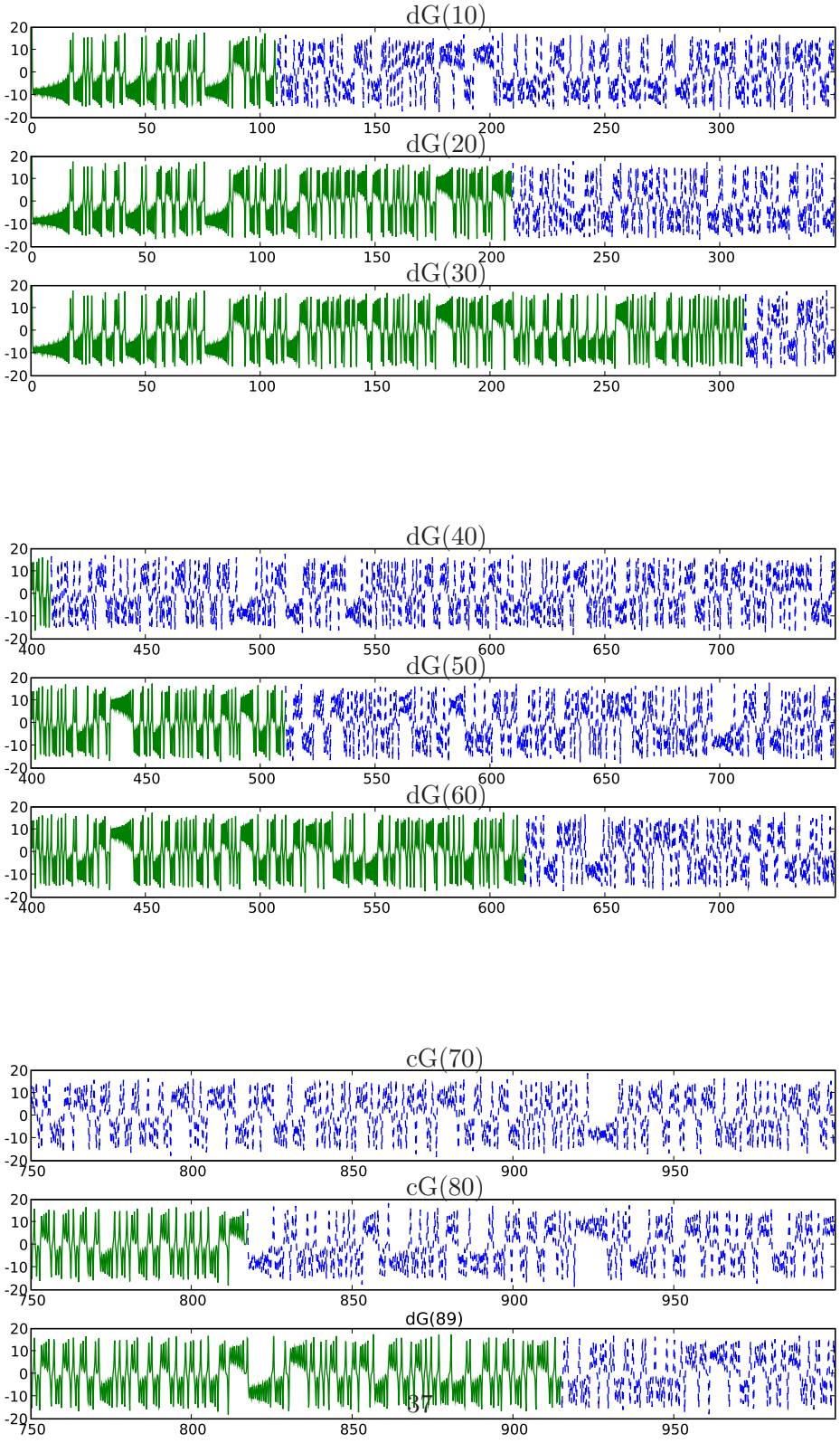


Figure 4.3: The full solution, componentwise.

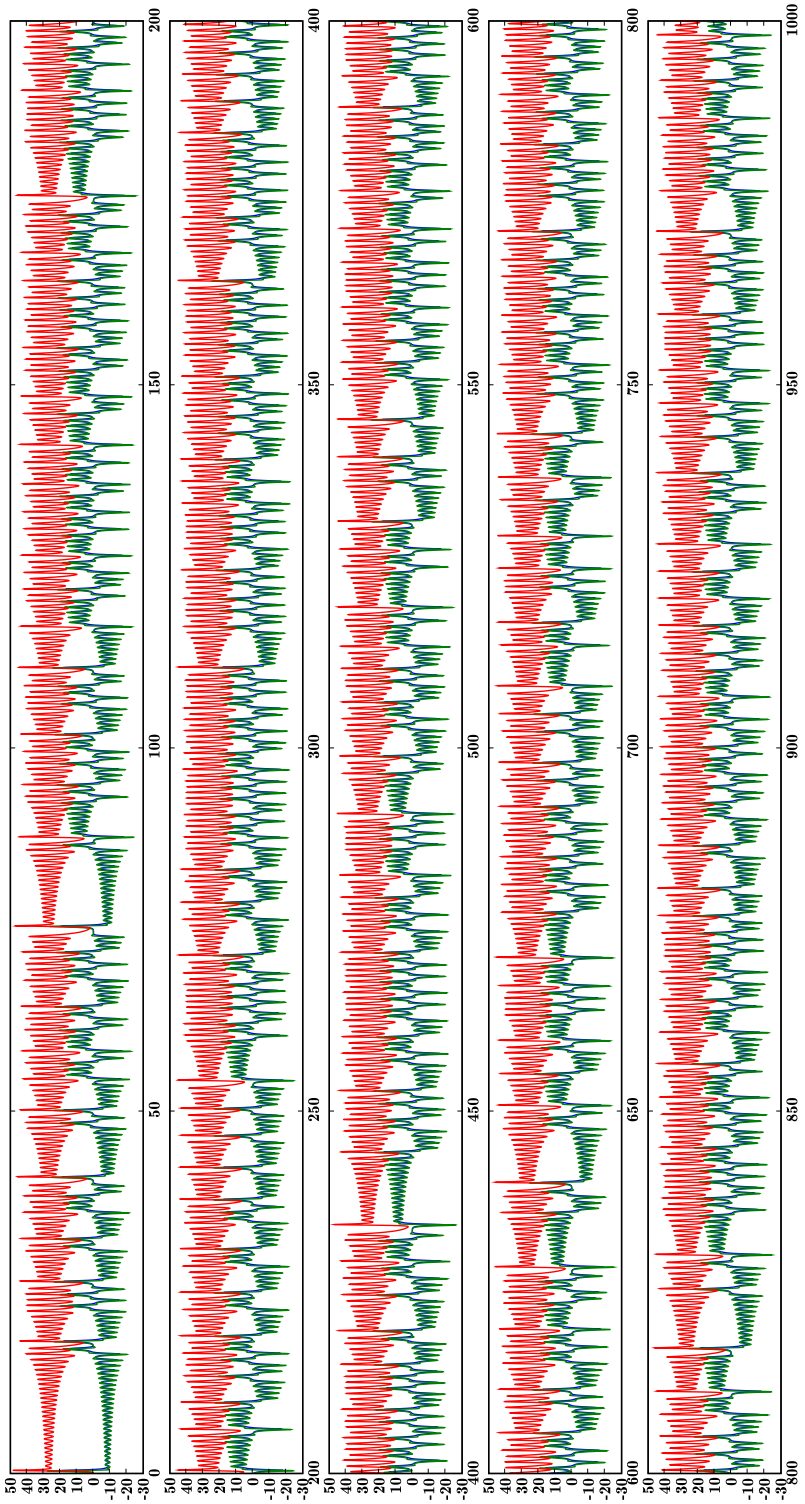


Figure 4.4: The trajectory of the full solution.

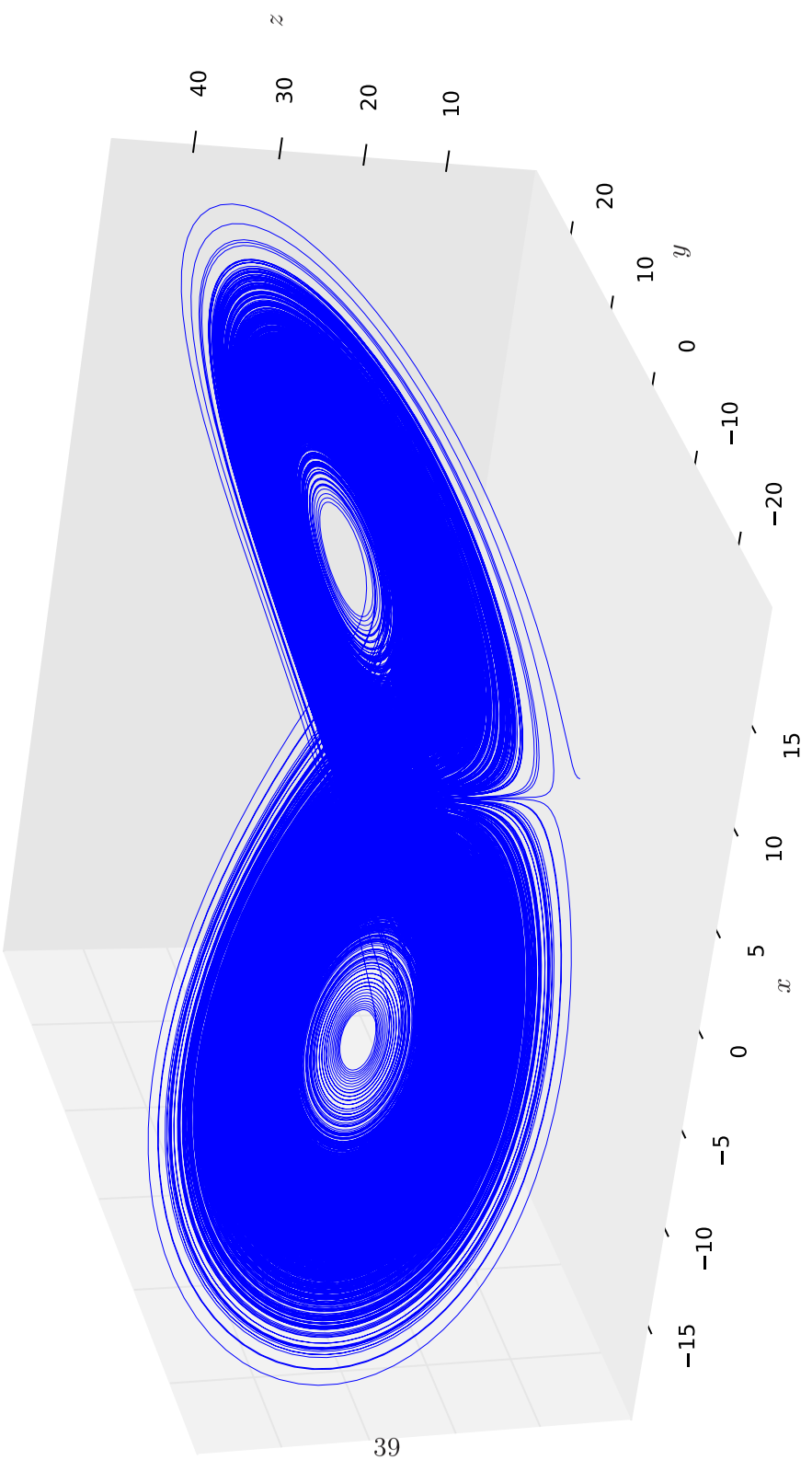


Figure 4.5: Absolute values of the linearized dual problem, z

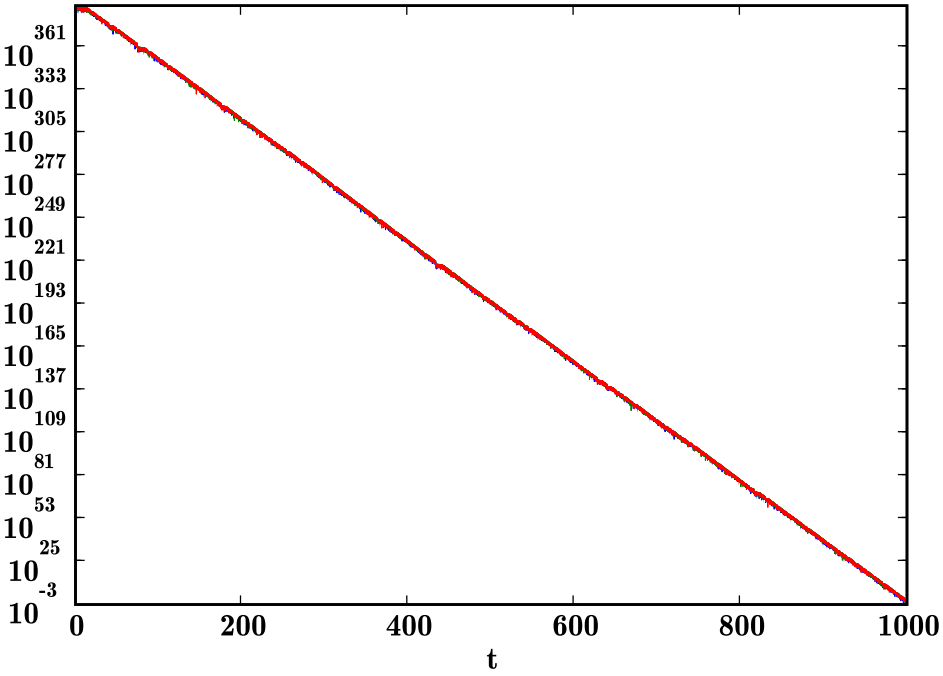


Figure 4.6: Part of the linearized dual problem, z

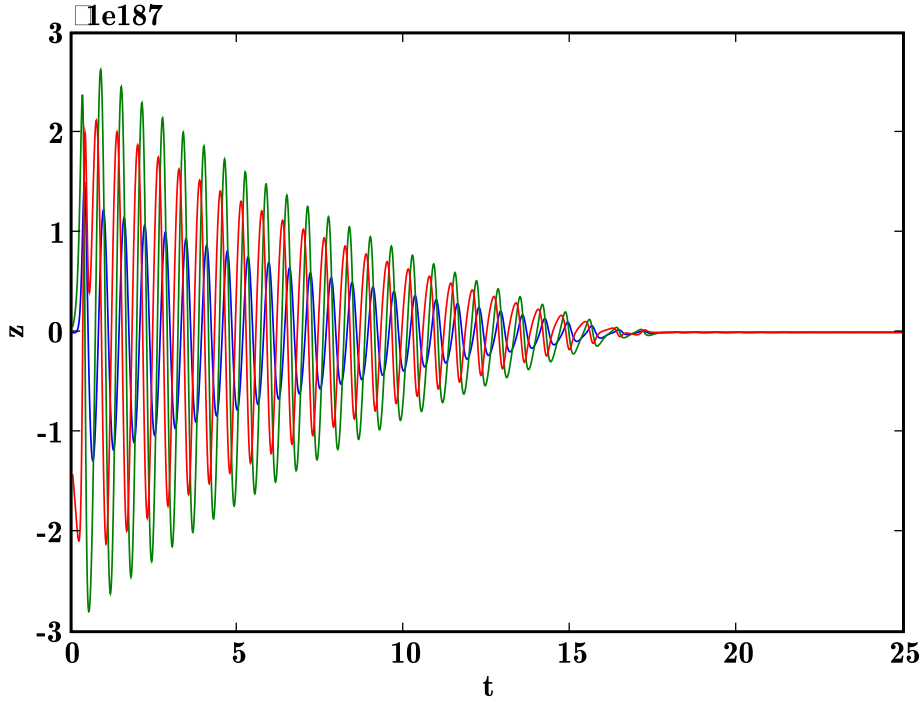


Figure 4.7: The Galerkin stability factor, S_G

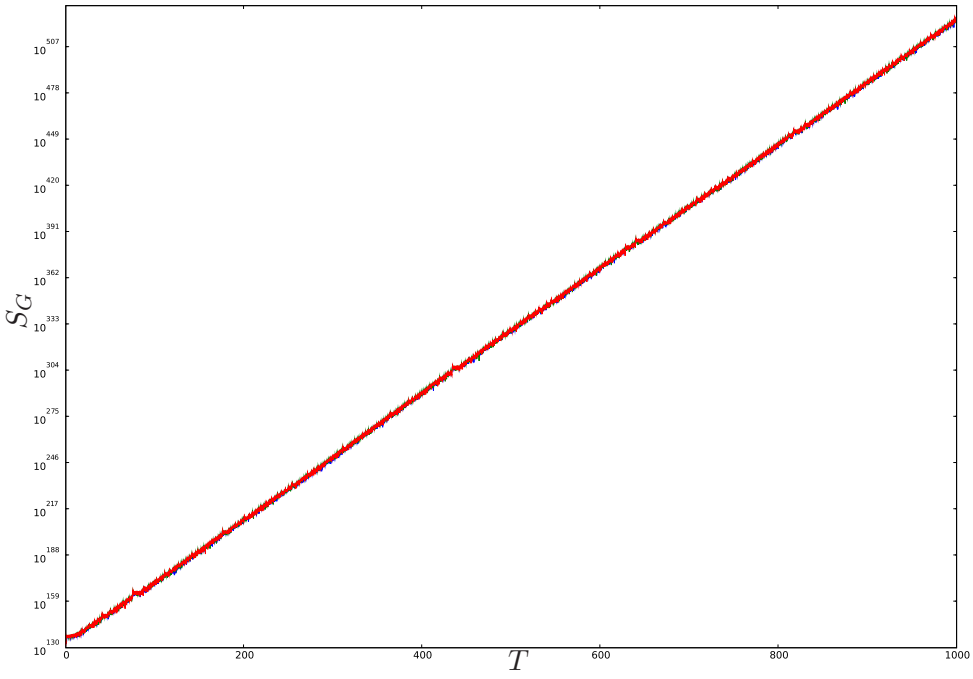


Figure 4.8: The computational stability factor, S_C with $q = 100$

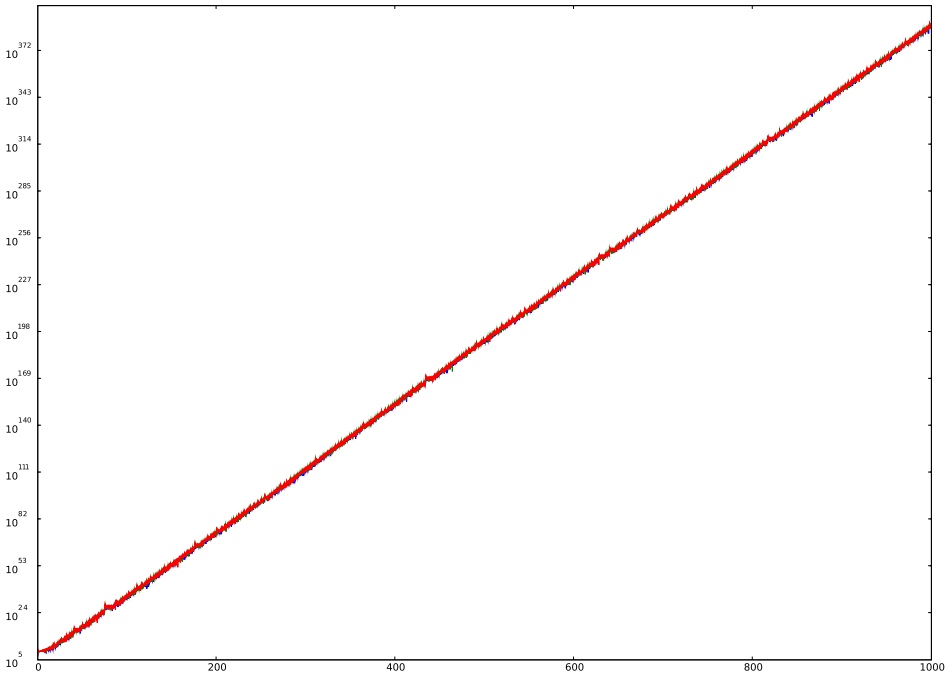


Figure 4.9: The x component of the computational stability factor S_C

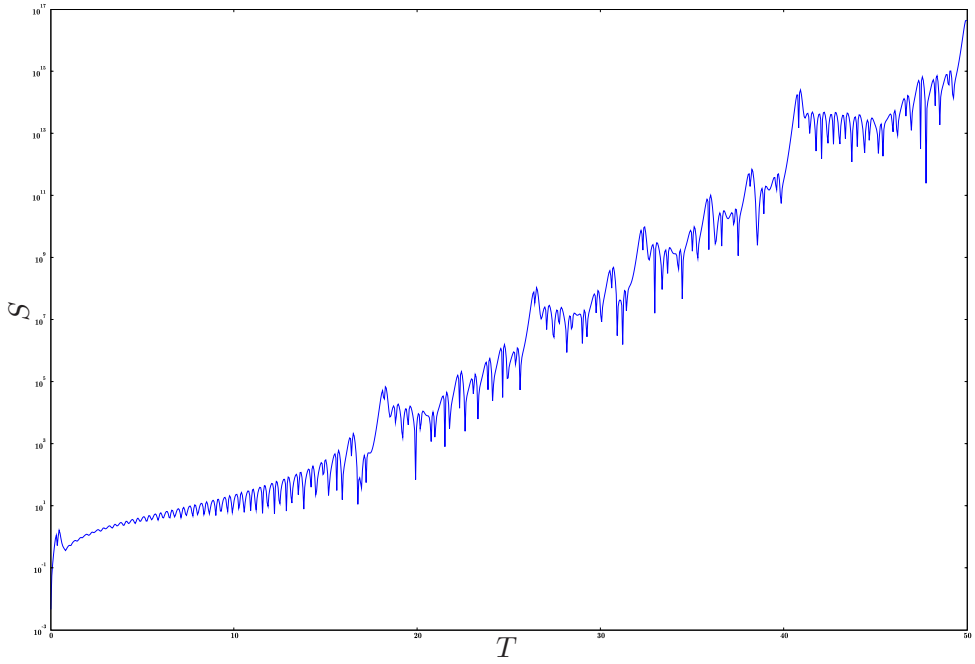


Figure 4.10: The accurate solution and a solution computed with cG(1)

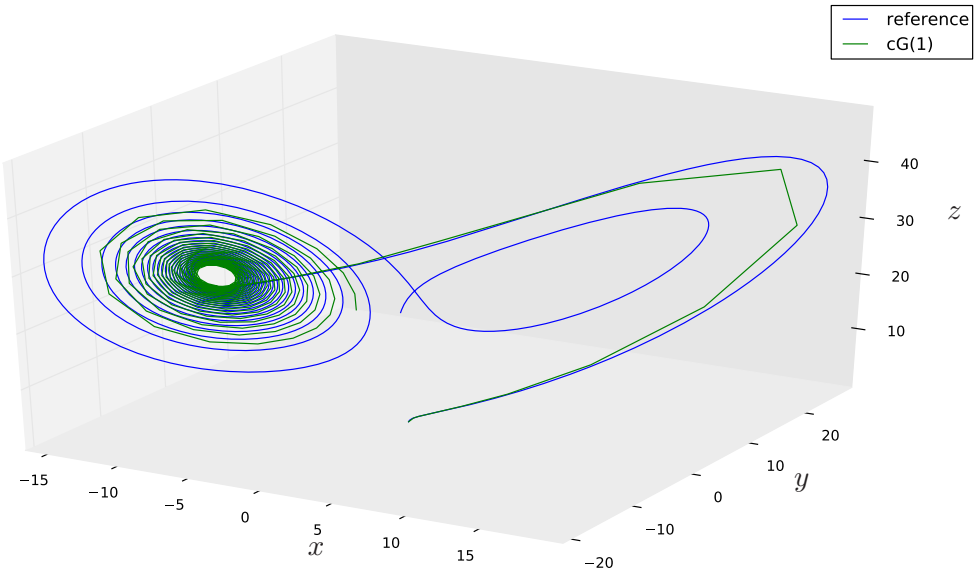
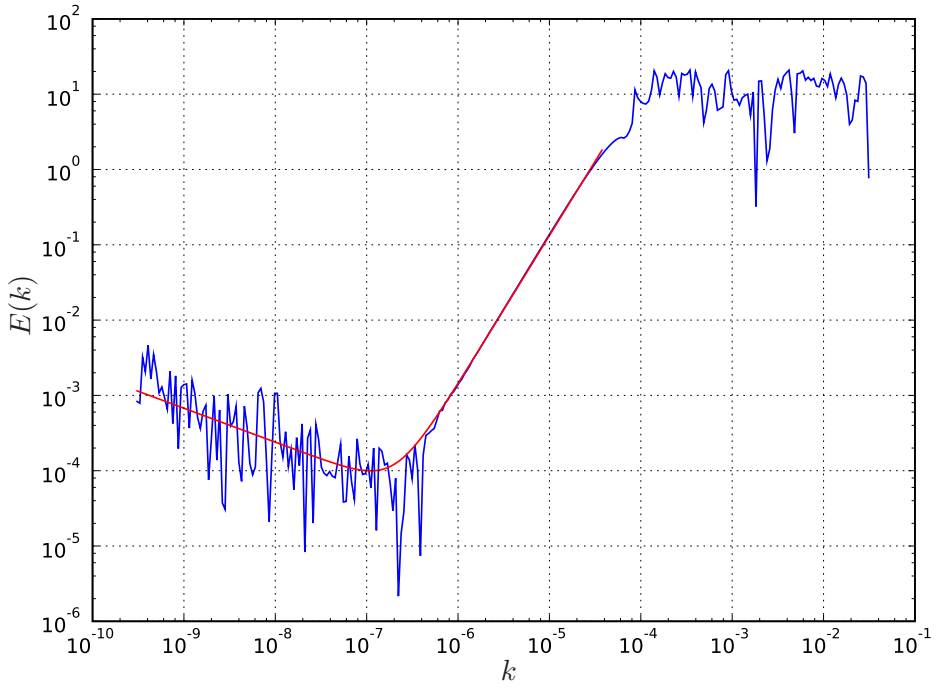


Figure 4.11: Error as function of k for the Lorenz system computed with cG(1)



Chapter 5

Discussion and concusion

5.1 Solutions of the Lorenz system

The fact that two different methods produce the same result obviously does not prove that the two soluations are accurate.

However from Tables 4.1 and 4.2 and Figures 4.1 and 4.2 we see clearly that the interval on which the solutions agree increases by approximately 100 when the order of the method is increased by 10. This pattern seems very regular. It it also common for the solutions computed with the cG(q) and dG(q) methods. Even if not proven, we are convinced that as cG(99) and cG(100) agree up to $t = 1016.0$, we have reached our goal of computing $u(1000)$.

5.1.1 Other results on the Lorenz system

In [1] Coomes, Kocak and Palmer compute so called pseudo orbits of the Lorenz system using a transversal shadowing method. They also show that their computed orbit is shadowed by a true orbit for a long period.

In [7] Jorba and Zou develop an ODE Solver called "Taylor". This solver takes a completely different approach compared to **DOLFIN**: Given an initial value problem of the form (1), it analytically differentiates the right hand side, $f(u, t)$, and outputs C code which can be compiled to build a timestepper for the specific problem. This timestepper uses the classic Taylor method. The solver chooses time step length and order adaptively. Taylor turns out to be very efficient for the Lorenz system and this solver is able to produce accurate solutions much faster than the ODE solver in **DOLFIN**. We have verified that the results produced with Taylor agree with our results up to $t = 1000$.

5.1.2 Objections to numerically computed solutions of attractors

We note that there is a certain controversy about using computed results of differential equations with chaotic behavior. In [21] Yao and Hughes reject the computability and argues that such computed solutions are "numerical noise" and show examples of solutions produced with different solvers and methods which all fail to converge. This paper is concerned about the so called "Lorenz winter system", but in [20] Yao argues similarly about 1.1. In [13] which is a reply to [21] E. Lorenz, however, argues that accurate solutions may be computed on a finite time interval and concludes

> In summary, numerical approximations can converge to a chaotic true solution throughout any finite range of time, although, if the range is large, confirming the convergence can be utterly impractical. If a uniformly convergent sequence of approximations is discovered, the true solution cannot be chaotic; seeking such a sequence is pointless, except perhaps as a test for chaos.

We will argue that our results support Lorenz's point of view. We have demonstrated convergence on longer and longer ranges of time by increasing the order of the method and reducing the time step size, while adjusting the floating-point precision accordingly, and paid the prize in terms of memory usage and, in particular, computation time. We also emphasize that the Taylor solver, totally independently and with a completely different method and approach produced the same result.

5.2 Stability factors

To be able to study the properties of the Lorenz equation, we want to observe how the stability factors evolve as the endtime T increases.

The stability factor is given by

$$S_q(T) = \int_0^T ||z^{(q)}|| \, \mathrm{d}t = \int_0^T ||w^{(q)}|| \, \mathrm{d}t$$

where w is the reversed dual problem as defined in (2.4).

Based on a number of computed stability factors computed with different q, and fitting them in a model thats reads

$$S_q(T) \approx 10^{\alpha q + \beta T + C}$$

with a least squares model, we obtain

$$S_q(T) \approx 10^{1.3q + 0.39T - 4.46}$$

Noting that $S_C = S^{[0]}(T)$, a simple model for the stability factor with respect to computational errors is then given by

$$S_C(T) \approx 10^{0.39T - 4.46}$$

which can be bounded by

$$S_C(T) \approx 10^{0.4T}.$$

This result indicates that, when computing a solution of the Lorenz system to $T = 1000$ the accumulated computational error can be estimated by

$$E_C \approx S_C \cdot \frac{\epsilon}{\min k} \approx 10^{385} \cdot \epsilon \tag{5.1}$$

Ignoring the time step length for now, ϵ must compensate for the stability factor. We conclude that it must be $\sim 10^{-385}$. This is also what we experinced when we computed solutions: 350 decimal digits precision was not enough to reach $T = 1000$. However, 400 digits precision was sufficient.

5.2.1 Approximating the u with U

When we computed numerical solutions of the dual problem, we were forced to approximate u with U assuming the difference is small. This seems inadequate. This difference is, after all, exactly what we are trying to estimate.

How sensitive the dual problem and the error estimate as a whole, is to such approximation is indeed an unanswered question in the field of a posteriori error analysis.

Does this mean our computations of the dual, and thereby the stability factors are worthless? We argue that they are not. We don't rely on the a posteriori analysis methods solely. We argue that our computed solution is correct mainly because solutions with different methods (and even different solvers, if we include Taylor) produce the same solution. The stability factors that we have computed using this approximation support these results. This supports, in our oppinion, both our solution and indicates that this kind of error estimates and compution of stability factors might not be very sensitive to this approximation of u, as long as U is close to u.

5.3 The error

In (4.1) we saw that the computational error as function of the timestep length (with constant T) seems to follow a model

$$E_C(k) \approx C \cdot k^{-0.45}$$

where C is constant. The corresponding analytical result, in (5.1), was that the error is modelled by

$$E_C(k) \approx C \cdot k^{-1}.$$

Note that both S_C and ϵ is included in C as they are constant with respect to k.

These two results do not conflict since we are dealing with estimates. Still, it raises the question of whether it is possible to obtain a sharper estimate of the computational error than (5.1) provides. We return to this issue in detail in a paper in preparation.

5.4 Conclusion

Built on the theory of a posteriori analysis by the dual problem we have implemented and domstrated methods for error analysis of ODEs.

All the tools used are publicly available through the ode solver in **DOLFIN** which as a side effect also has been optimized.

Our goal was to compute $u(1000)$ of (1.1) and to verify the result with different techniques. We conclude that we have achieved this. The result is supported by both our relatively simple comparisons, the slightly more sophisticated analysis and by an independent solver. We have demonstrated Edvard Lorenz's point: It is possible to compute solutions of a chaotic system if you're willing to pay the price in terms of computation.

5.5 Future work

During this project we have developed tools for stability analysis and error estimation. Still some work work remains, before the process of computing a solution, verified to be within the given tolerance, is completely automated.

The experience with the Taylor solver showed that generating code can be very efficient. In fact, the PDE solver in **DOLFIN** takes a similar approach even if the methods are completely different. Investigating whether such an approach can be useful within the framework of Galerkin methods is interesting.

As mentioned in Section 5.3, there are unanswered questions, in the field of a posteriori error analysis, in particular regarding the linearization error we made approximating u by U. We will look closer into some of them soon.

Bibliography

[1] Brian A. Coomes, Hüseyin Koçak, and Kenneth J. Palmer. Rigorous computational shadowing of orbits of ordinary differential equations. *Numer. Math.*, 69(4):401–421, 1995.

[2] E Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numerica*, pages 105–158, 1995.

[3] K. Eriksson, D. Estep, and C. Johnson. *Applied Mathematics Body and Soul, Volume 3*. Springer, Berlin, 2003.

[4] Don Estep and Claes Johnson. The pointwise computability of the lorenz system. *Mathematical Models and Methods in Applied Science*, 8(8):1277–1305, 1998.

[5] FEniCS. FEniCS project. URL: `http//www.fenics.org/`.

[6] GMP, 2010. `http://www.gmplib.org/`.

[7] Àngel Jorba and Maorong Zou. A software package for the numerical integration of ode by means of high-order taylor methods. *Exprimental Mathematics*, 14:99–117, 2005.

[8] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010. To appear.

[9] A. Logg, G. N. Wells, et al. DOLFIN. URL: `http//www.fenics.org/dolfin/`.

[10] Anders Logg. Multi-adaptive galerkin methods for odes ii: Applications. *Chalmers Finite Element Center Preprint*, 1(10), 2001.

[11] Anders Logg. Multi-adaptive galerkin methods for odes i. *SIAM J. Sci. Comput.*, 24(6):1879–1902, 2002.

[12] Anders Logg. Multi-adaptive galerkin methods for odes ii: implementation and applications. *SIAM J. Sci. Comput.*, 25(4):1119–1141, 2003.

[13] Edvard N. Lorenz. Reply to comment by l.-s. yao and d. hughes. *Tellus A*, 60(4):806–807, 2008.

[14] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, 1963.

[15] MPFR, 2010. `http://www.mpfr.org/`.

[16] R. B. Sidje. EXPOKIT. A software package for computing matrix exponentials. *ACM Trans. Math. Softw.*, 24(1):130–156, 1998.

[17] Steve Smale. Mathematical problems for the next centur. *Math. Intelligencer*, 20(2):7–15, 1998.

[18] Warwick Tucker. The lorenz attractor exists. *C. R. Acad. Sci. Paris*, 328:1197–1202, 1999.

[19] Marcelo Viana. What's new on lorenz strange attractors? *Mathematical Intelligencer*, 22(3):6–19, 2000.

[20] Lun-Shin Yao. What is new on computed lorenz strange attractors: chaos or numerical errors? *ArXiv Mathematics e-prints*, May 2003.

[21] Lun-Shin Yao and Dan Hughes. Comment on "computational periodicity as observed in a simple system," by edward n. lorenz (2006a). *Tellus A*, 60(4):803–805, 2008.